Testplanung:

Testfallentwurf und Modultests

Großes Seminar bei
Prof. Dr. K. Wüst
und
Prof. Dr. P. Kneisel

von
Tim Aßmann

This document is about Test-Case Design and Module Testing. The first part of the essay describes two important testing strategies which are used to find appropriate test cases: Whitebox-Testing and Blackbox-Testing. By using the Whitebox-Testing methods, you analyze the source code of a program and use this knowledge for finding suitable test data. The Blackbox-Testing-Methods work different. Here you don't know the source code, but only the input and output specifications. So you create test cases by using different methods based on the input and output data.

The second part of this document points out what has to be considered when testing a module based program. For this the different test strategies are either incremental or non-incremental testing. The advantages and disadvantages of each method are described in this chapter of the document.

Inhaltsverzeichnis

Einleitung

Jeder, der schon einmal Software testen musste, wird auf die Schwierigkeit gestoßen sein, geeignete Testdaten zu finden. Oft wird hierauf nicht genug Sorgfalt gelegt. Die Praxis sieht leider oftmals so aus, dass die Prüfer sich keine großen Gedanken machen und mehr oder weniger wild Testdaten erfinden. Die dabei entstehenden Testfälle sind jedoch eher zufällige Eingabedaten und die Wahrscheinlichkeit, dass diese Testdaten zu einem guten Testergebnis führen (also möglichst viele Fehler finden), ist eher gering. Natürlich können auch so Fehler aufgedeckt werden, jedoch wird man hier wohl kaum eine gute Ausbeute erlangen.

Erfahrene Tester werden auf Methoden zurückgreifen, um sinnvolle Testdaten zu finden. Mit „sinnvoll“ ist gemeint, dass man mit möglichst wenigen Testfällen möglichst viele Fehler aufdeckt. Dies ist aus Kostengründen empfehlenswert, denn Testfälle durchzuführen und auszuwerten kostet Zeit und somit Geld und jeder Fehler, den man in der Testphase nicht findet, wird später ein Vielfaches kosten, wenn dieser nach der Auslieferung auftritt.

Inhalt dieses Seminars ist es, geeignete Maßnahmen vorzustellen, um diese sinnvollen Testdaten gezielt zu finden. Dabei werden diese Methoden in zwei übergreifende Bereiche eingeteilt: Whitebox- und Blackbox-Tests.

Im Anschluss daran wird noch einmal speziell auf Modultests eingegangen. Die im ersten Teil dieses Dokumentes beschriebenen Methoden eignen sich größtenteils für kleinere Code-Abschnitte und sind nur bedingt auf komplette Anwendungen übertragbar. Außerdem sind größere Programme in der Regel auch schon durch strukturierte Softwareplanung in kleinere Module bzw. Klassen unterteilt. Daher wird man in der Praxis häufig Module zu testen haben. Die dabei zu beachtenden speziellen Vorgehensweisen zum Testen von Modulen ist Inhalt des zweiten Teils dieses Seminars.

Testfallentwurf

Die Methoden, um sinnvolle Testdaten zu finden, unterscheiden sich in zwei übergreifende Bereiche: Whitebox-Tests und Blackbox-Tests. Während man bei dem Whitebox-Test die interne Struktur eines Programms unter Berücksichtigung und Vorlage des Quelltextes analysiert und auf dieser Basis seine Testfälle entwirft, sieht man beim Blackbox-Test das Programm oder Modul wie der Name schon sagt als so genannte Blackbox an. Das heißt, dass man hier den Quell-Code nicht berücksichtigt, sondern man versucht, alleine aufgrund von Ein- und Ausgabespezifikationen geeignete Testdaten zu finden und das „Innenleben“ des Programms ausgeblendet wird. Bei einer umfassenden Testplanung sollten beide Methoden berücksichtigt werden.

Whitebox-Test

Wie oben schon erwähnt, analysiert man hier die zu testende Software unter Berücksichtigung des Quelltextes. Als erstes macht man den Steuerfluss des Programms sichtbar und kann daraufhin verschiedene Methoden anwenden, um Testfälle zu finden.
In Abb.1 sieht man z.B. den Steuerfluss einer Schleife, die bis zu 20-mal ausgeführt werden kann. Innerhalb dieser Schleife gibt es 5 verschiedene Pfade. Die Knotenpunkte sind Abfolgen von Befehlen, die Verbindungen zeigen die Reihenfolge an, in denen diese Codeabschnitte abgearbeitet werden können. Bei den Aufspaltungen handelt es sich hier um „if-Abfragen“.
Als erstes soll hier der Sinn von ausgewählten Testfällen gezeigt werden und dass vollständiges (erschöpfendes) Testen nicht möglich ist. Würde man vollständig Testen wollen, so müsste man jeden möglichen Pfad des Programms mit Hilfe eines Testfalles abarbeiten. Dies wären bei einmaligem Durchlaufen der Schleife 5 Testfälle. Bei zweimaligem Durchlaufen wären das schon $5 \times 5 = 5^2$ Testfälle, usw. Daraus würden sich bei bis zu 20 Durchläufen $5 + 5^2 + 5^3 + \ldots + 5^{20} \approx 119$ Billionen Möglichkeiten ergeben.
Um es noch deutlicher zu machen, dass dadurch vollständiges Testen unmöglich ist, möchte ich folgendes hinzufügen: Würde man jede Minute einen Testfall entwerfen, umsetzen und auswerten, so wäre man ca. 225 Millionen Jahre beschäftigt. Dass das etwas lang ist, wird

vermutlich niemand bestreiten. Daher muss man das Ganze durch einzelne stichhaltige Testfälle eingrenzen.

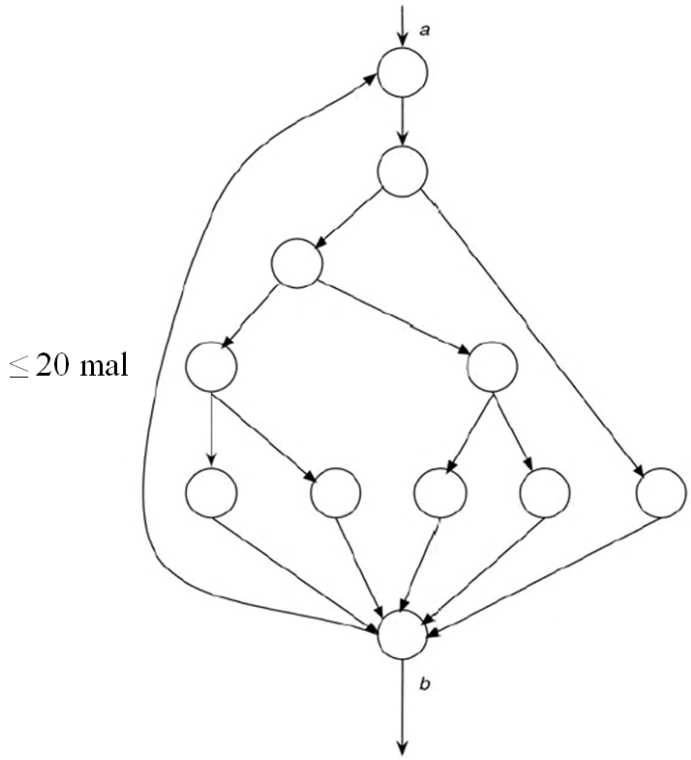

Abbildung 1: Steuerfluss einer Schleife

Außerdem möchte ich anmerken, dass selbst ein vollständiger Whitebox-Test keine Fehlerfreiheit eines Programms garantieren könnte. So würde eine falsche Spezifikation durch die Analyse des Quelltextes nicht auffallen. Auch komplette fehlende Pfade würden durch diese Analyse nicht unbedingt auffallen.

Zur Veranschaulichung werde ich die Whitebox-Methoden anhand eines Beispieles möglichst praxisnah behandeln. Der Quelltext zu diesem Beispiel ist in Abb.2 zu sehen. Dort sieht man eine kleine Funktion foo(), die zwei „if“-Abfragen enthält. Aus diesem Quelltext kann man den Steuerfluss grafisch erstellen. Dies ist in Abb.3 zu sehen. Die einzelnen Pfade sind mit den Buchstaben „a“ bis „e“ markiert, um den Steuerfluss der einzelnen Testfälle deutlich zu machen.

```
public void foo(int a, int b, int x) {
    if (a>1 && b==0)  {
        x=x/a;
    }
    if (a==2 || x>1)  {
        x=x+1;
    }

}
```

Abbildung 2: Quelltext einer kleinen Funktion

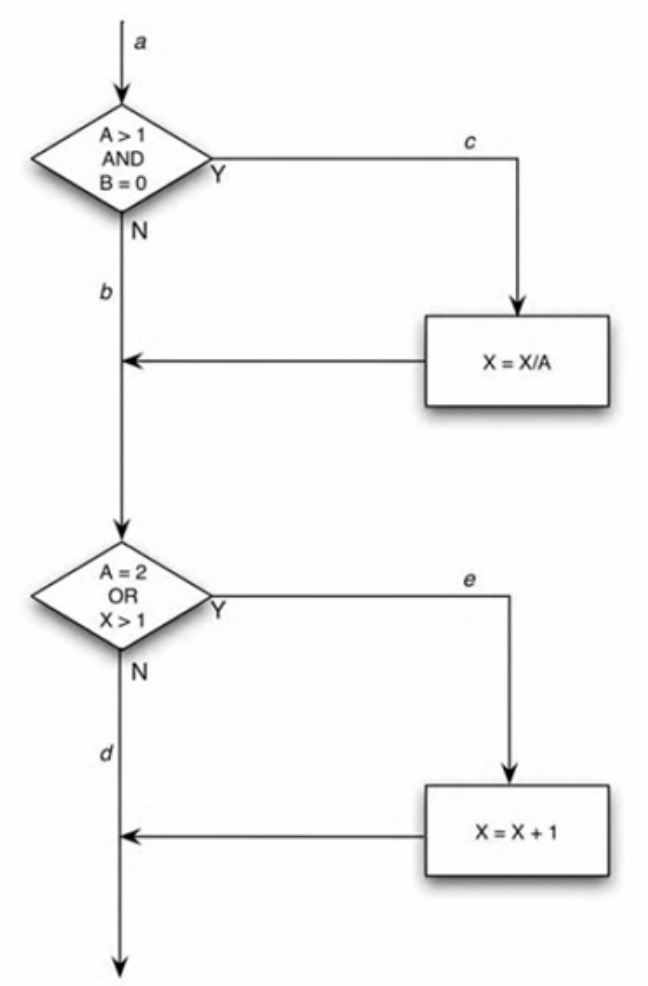

Abbildung 3: Steuerfluss des Programms aus Abb.2

Ausführung aller Befehle

Die erste Idee, die man bei der Eingrenzung von Testfällen hat, ist, dass jeder Befehl mindestens einmal ausgeführt werden muss. Dies würde bei diesem Beispiel durch einen Testfall, der über den Pfad „ace“ geht, abgedeckt sein. Setzt man also die Werte **A = 2, B = 0 und X = 4** ein, so wird das erste Befehlssegment (X = X / A) und auch das zweite Befehlssegment (X = X + 1) ausgeführt.

Wie man sich sicherlich denken kann, ist diese Methode nicht sehr effektiv beim Auffinden von Fehlern. Nichts desto trotz ist es sinnvoll, dass jeder ausführbare Befehl einmal durchlaufen wurde. Dies ist also bei den nachfolgenden Methoden immer als Mindestanforderung zu berücksichtigen.

Zur weiteren Einschätzung der Methoden habe ich ein paar Fehlerfälle erstellt und werde die Whitebox-Methoden unter Berücksichtigung dieser Fehlerfälle untersuchen, um deutlich zu machen, wie diese Methoden einzuschätzen sind. Natürlich kann ich im Rahmen dieser Untersuchung nur eine kleine Auswahl an Fehlern betrachten, da eine statistisch aussagekräftigere Untersuchung den Rahmen dieses Seminars sprengen würde. Die Grundzüge sollten allerdings klar werden. Für die „Ausführung aller Befehle“ sind diese Fehlerfälle, und ob sie durch den oben genannten Testfall entdeckt würden, in Abb.4 zu sehen.

Mögliche Fehler	Befehle	Entsch.	Bed.	Ent./ Bed.	Mehrf. -Bed.
Bei erfolgr. 2. If-Bedingung: x = x – 1	✓				
Pfad d sollte es nicht geben	x				
1. If sollte auf || statt auf && prüfen	x				
2. If sollte auf && statt auf || prüfen	x				
2. If sollte auf x ≥ 1 statt x > 1 prüfen	x				
2. If sollte auf x < 1 statt x > 1 prüfen	x				

Abbildung 4: Fehlerfälle

Wie man hier sieht, wird nur – wie schon oben erwähnt – der Fehler, welcher sich auf ein Stück ausführbaren Codes bezieht, gefunden. Sollte oben in dem Codebeispiel z.B. statt X = X + 1 innerhalb der zweiten „if“-Bedingung X = X – 1 stehen, so würde dieses durch den

oben definierten Testfall aufgedeckt werden. Alle anderen Fehler jedoch würden nicht erkannt. Sollte es z.B. den Pfad „d“ nicht geben, bei dem jedoch kein Code ausgeführt würde, so käme dies mit der Testmethode nicht heraus. Die Folge daraus wäre ein sehr schlechter Test, da die meisten Fehler nicht gefunden würden. Vor allem Fehler bei Entscheidungen sind so nur eher zufällig zu entdecken. Dies bringt uns zur nächsten Idee, der „Ausführung aller Entscheidungen“.

Ausführung aller Entscheidungen

Die Methode bei der „Ausführung aller Entscheidungen“ stützt sich auf der Idee, dass man bei Bedingungen nicht nur den Pfad einschlägt, der zu ausführbarem Code führt, sondern man für jede Entscheidung einmal alle möglichen direkten weiteren Pfade abarbeitet. Das bedeutet z.B. konkret zu dem oben genannten Beispiel, dass man hier nicht nur die Pfade „c“ und „e“ bei den jeweiligen Entscheidungen einschlägt, sondern auch Testfälle entwirft, die in die Pfade „b“ und „d“ führen.

Hier kann man nun entweder Testfälle erstellen, die die Pfade „abd“ und „ace“ abarbeiten oder die über die Pfade „acd“ und „abe“ führen. Somit hat man bei jedem „if“ einmal den „Ja“-Zweig (Bedingung erfüllt) und einmal den „Nein“-Zweig (Bedingung nicht erfüllt) der „if“-Statements eingeschlagen.

Die folgenden beiden Testfälle erfüllen diese Voraussetzungen:

1) **A = 3, B = 1, X = 0 (abd)**
2) **A = 2, B = 0, X = 4 (ace)**

Betrachten wir nun, inwiefern die oben entworfenen Fehlerquellen aufgedeckt werden:

Mögliche Fehler	Befehle	Entsch.	Bed.	Ent./ Bed.	Mehrf. -Bed.
Bei erfolgr. 2. If-Bedingung: x = x – 1	✓	✓			
Pfad d sollte es nicht geben	x	✓			
1. If sollte auf || statt auf && prüfen	x	✓			
2. If sollte auf && statt auf || prüfen	x	x			
2. If sollte auf x ≥ 1 statt x > 1 prüfen	x	x			
2. If sollte auf x < 1 statt x > 1 prüfen	x	✓			

Abbildung 5: Fehlerfälle

Wie man hier sehen kann, werden durch diese Testfälle schon mehr Fehler aufgedeckt, jedoch ist auch diese Methode noch sehr ungenügend. Das bringt uns zur nächsten Methode, der „Ausführung aller Bedingungen“.

Ausführung aller Bedingungen

Bei der Ausführung aller Bedingungen wird im Gegensatz zu der vorherigen Methode nicht nach „Ja“-Zweig und „Nein“-Zweig der „if“-Abfragen (Entscheidungen) geschaut, sondern man geht tiefer ins Detail und betrachtet Testfälle, die sich auf die vier Bedingungen ($A > 1$, $B = 0$ und $A = 2$, $X > 1$) beziehen. Man sucht nun Testfälle, die diese Bedingungen einmal erfüllen und einmal nicht erfüllen. Dadurch ergeben sich folgende Testfälle:

- Testfälle für $A > 1$, $A \leq 1$ und $B = 0$, $B \neq 0$ (erstes „if“).
- Testfälle für $A = 2$, $A \neq 2$ und $X > 1$, $X \leq 1$ (zweites „if“).

Zu beachten hierbei ist, dass die Bedingung $X > 1$, $X \leq 1$ beim zweiten „if“ anliegen muss und X evtl. schon nach dem ersten „if“ verändert sein kann, nämlich wenn der definierte Testfall über den „c“-Pfad zu dem zweiten „if“ gekommen ist.

Folgende Testdaten erfüllen diese Voraussetzungen:

1) **A = 1, B = 0, X = 4 (abe)**
2) **A = 2, B = 1, X = 1 (abe)**

Diese Testfälle decken alle oben erwähnten Bedingungen ab, allerdings führen diese beiden über den Pfad „abe“. Daraus folgt, dass die Anweisung, welche über den Pfad „c“ erreicht werden kann, nicht ausgeführt wird. Da wir allerdings anfangs bei der Betrachtung der „Ausführung aller Befehle“ festgelegt haben, dass jeder Befehl mindestens einmal ausgeführt werden soll, muss hier noch ein weiter Testfall definiert werden:

3) **A = 2, B = 0, X = 4 (ace)**

Nun wird auch der Teilpfad „c“ abgearbeitet. In Abb.6 ist die daraus resultierende Beispiel-Fehlertabelle zu sehen.

Wie man unschwer erkennen kann, werden zwar auch mit dieser Methode einige Fehler aufgedeckt, aber andererseits ist auch diese noch verbesserungsfähig. Wie man nämlich erkennen kann, wird bei den drei Testfällen der Pfad „d“, bei dem jedoch kein ausführbarer Code steht, nicht abgedeckt. Dieser wurde mit der vorherigen Methode jedoch gefunden.

Dadurch kommt die Idee auf, diese beiden Methoden zu kombinieren und somit noch bessere Testdaten zu ermöglichen.

Mögliche Fehler	Befehle	Entsch.	Bed.	Ent./ Bed.	Mehrf. -Bed.
Bei erfolgr. 2. If-Bedingung: x = x – 1	✓	✓	✓		
Pfad d sollte es nicht geben	x	✓	x		
1. If sollte auf || statt auf && prüfen	x	✓	✓		
2. If sollte auf && statt auf || prüfen	x	x	✓		
2. If sollte auf x ≥ 1 statt x > 1 prüfen	x	x	x		
2. If sollte auf x < 1 statt x > 1 prüfen	x	✓	✓		

Abbildung 6: Fehlerfälle

Ausführung aller Entscheidungen/Bedingungen

So kommt man zu der Methode der Ausführung aller Entscheidungen/Bedingungen, welche die beiden vorhergehenden Vorgehensweisen kombiniert. Man entwirft wie im letzten Abschnitt Testfälle, die die 4 Bedingungen berücksichtigen, achtet jedoch auch darauf, dass auch alle „Entscheidungen“ beachtet werden. Die folgenden beiden Testfälle erfüllen diese Voraussetzungen:

1) **A = 2, B = 0, X = 4 (ace)**
2) **A = 1, B = 1, X = 1 (abd)**

Die daraus folgende Fehlertabelle sieht folgendermaßen aus:

Mögliche Fehler	Befehle	Entsch.	Bed.	Ent./ Bed.	Mehrf. -Bed.
Bei erfolgr. 2. If-Bedingung: x = x – 1	✓	✓	✓	✓	
Pfad d sollte es nicht geben	x	✓	x	✓	
1. If sollte auf || statt auf && prüfen	x	✓	✓	x	
2. If sollte auf && statt auf || prüfen	x	x	✓	x	
2. If sollte auf x ≥ 1 statt x > 1 prüfen	x	x	x	✓	
2. If sollte auf x < 1 statt x > 1 prüfen	x	✓	✓	x	

Abbildung 7: Fehlerfälle

Anders als zunächst erwartet, gibt es auch hier noch viele Fehler, die nicht entdeckt werden. Wie anfangs erwähnt, reicht die Menge an hier behandelten „Fehlerfällen“ natürlich nicht aus, um ein statistisch repräsentatives Bild aufzuzeigen, außerdem grenzen diese Methoden die Findung der Testfälle nur ein und es ist immer noch Spielraum beim Erstellen der Testfälle vorhanden, da ja nur Rahmenbedingungen vorgegeben werden. Aber in der Regel ist die Methode „Ausführung aller Entscheidungen/Bedingungen“ jedoch den vorhergehenden vorzuziehen.
Durch nähere Betrachtung des Steuerflusses und weitere Aufschlüsselung findet man allerdings eine noch bessere Methode, die „Ausführung aller Mehrfachbedingungen“.

Ausführung aller Mehrfachbedingungen

Die Frage, die sich stellt ist, warum auch bei der vorherigen Methode immer noch so viele Fehler nicht aufgedeckt werden. Bei näherer Betrachtung wird klar werden, dass einige Bedingungen andere Bedingungen maskieren. Dazu untersuchen wir, wie ein Compiler diesen Steuerfluss abarbeiten könnte. Dieser würde die beiden „if“-Abfragen noch weiter aufschlüsseln, wodurch „versteckte“ Pfade sichtbar werden (siehe Abb.8).

Greift man jetzt noch mal auf die Testfälle aus dem vorherigen Unterkapitel (Ausführung aller Entscheidungen/Bedingungen) zurück, so kann man nun erkennen, dass nicht, wie man vorher dachte, alle Entscheidungen/Bedingungen berücksichtigt werden. Die Bedingung $B = 0$ wird von der Bedingung $A > 1$ maskiert. Wendet man nun nämlich die erwähnten Testfälle auf diesen aufgeschlüsselten Steuerfluss aus Abb.8 an, so wird man erkennen, dass der „Nein“-Pfad bei der Bedingung $B = 0$ nicht eingeschlagen wird. Unter diesen Gesichtspunkten kommt man zu einer weiteren Methode, der „Ausführung aller Mehrfachbedingungen“. Hierbei werden alle möglichen Bedingungskombinationen für jedes „if“ betrachtet.

Das erste „if“ liefert die folgenden Bedingungen:

1) $A > 1, B = 0$
2) $A > 1, B \neq 0$
3) $A \leq 1, B = 0$
4) $A \leq 1, B \neq 0$

Aus dem zweiten „if“ leiten sich hingegen folgende Bedingungen ab:

5) $A = 2, X > 1$
6) $A = 2, X \leq 1$
7) $A \neq 2, X > 1$
8) $A \neq 2, X \leq 1$

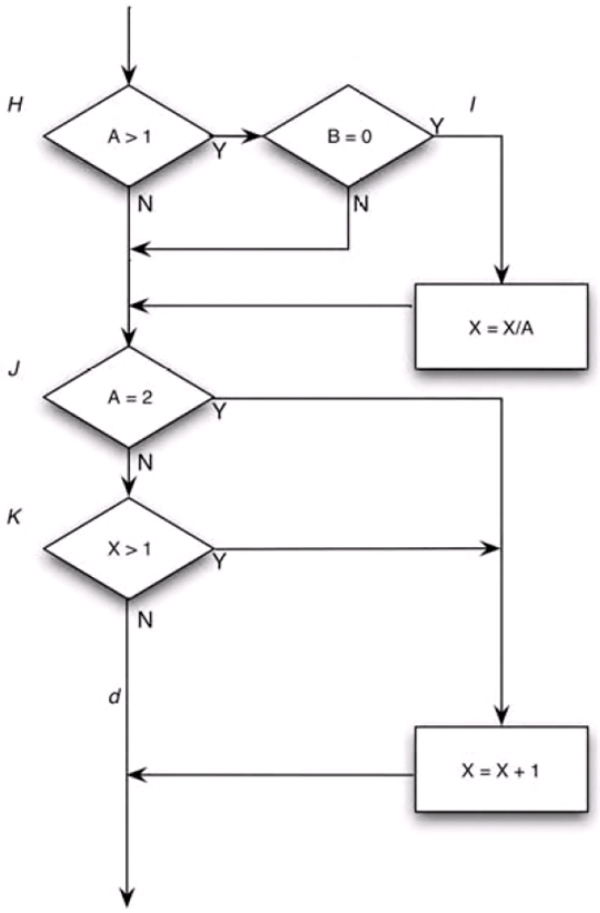

Abbildung 8: Ausführlicher Steuerfluss des Programms aus Abb.2

Nimmt man diese Voraussetzungen – wobei man auch hier wieder beachten muss, dass die Bedingung X beim zweiten „if“ zu gelten hat und nach dem ersten „if“ schon verändert sein kann – so kann man folgende Testfälle wählen, welche diese Bedingungen alle erfüllen:

1) **A = 2, B = 0, X = 4 (ace)** (aus 1) und 5))
2) **A = 2, B = 1, X = 1 (abe)** (aus 2) und 6))

3) **A = 1, B = 0, X = 2 (abe)** (aus 3) und 7))
4) **A = 1, B = 1, X = 1 (abd)** (aus 4) und 8))

Hier haben wir also 4 Testfälle, wobei man darauf achten sollte, dass diese 4 Testfälle nicht gleich der „Ausführung aller Pfade“ sind. Der Pfad „acd“ wird nämlich nicht abgearbeitet. Bei diesem Beispiel haben wir auch nur zufällig 4 Testfälle, was gleich der Anzahl der Testfälle ist, die man hier bei der Ausführung aller Pfade hätte. Sobald jedoch eine Schleife in einem Programm vorhanden ist, wird spätestens deutlich, dass die Zahl der Testfälle bei dieser Methode nicht exponentiell steigt, wie das bei der Ausführung aller Pfade der Fall wäre.
Nun fehlt noch die Kontrolle der Fehlerfälle:

Mögliche Fehler	Befehle	Entsch.	Bed.	Ent./ Bed.	Mehrf. -Bed.
Bei erfolgr. 2. If-Bedingung: x = x – 1	✓	✓	✓	✓	✓
Pfad d sollte es nicht geben	x	✓	x	✓	✓
1. If sollte auf || statt auf && prüfen	x	✓	✓	x	✓
2. If sollte auf && statt auf || prüfen	x	x	✓	x	✓
2. If sollte auf x ≥ 1 statt x > 1 prüfen	x	x	x	✓	✓
2. If sollte auf x < 1 statt x > 1 prüfen	x	✓	✓	x	✓

Abbildung 9: Fehlerfälle

Hieraus wird wohl unweigerlich deutlich, dass diese Methode die sinnvollste ist, um per Whitebox-Analyse Testfälle zu finden. Allerdings ist es auch die ausführlichste Methode und diese wird bei größeren Programmabschnitten sehr schnell sehr aufwendig.
Was an dieser Stelle noch einmal erwähnt werden sollte ist, dass auch mit dieser Methode nicht zwangsweise alle Fehler gefunden werden. Die hier ausgewählten Fehlerfälle werden zwar abgedeckt, aber dies ist keine Garantie, dass man alle Fehler findet – die Wahrscheinlichkeit, qualitativ hochwertige Testdaten zu finden, ist jedoch deutlich höher, als wenn man zufällige Testdaten zum Testen heranzieht.
Um das Ganze noch einmal zu unterstreichen, dass nicht zwangsweise alle Fehler mit dieser Methode gefunden werden, soll folgendes Beispiel aufzeigen: „Ein Programm soll testen, ob die Differenz von 2 Zahlen kleiner einer dritten Zahl ist.“

Erstellt nun jemand daraufhin folgendes (fehlerhaftes!) Programm „If (a – b < c) …;“ und testet man dieses mit der letzten (besten) Whitebox-Methode, so könnte man zu folgenden Testdaten kommen:

1) a = 2, b = 1, c = 3 ==> true (wie erwartet)
2) a = 5, b = 1, c = 3 ==> false (wie erwartet)

Man sieht also, dass diese Testdaten ergeben, dass kein Fehler in dem Programm vorliegt, oder besser, kein Fehler gefunden wurde. Betrachtet man jedoch den Fall, dass a = 1, b = 5 und c = 3 ist, so liefert das Programm fälschlicherweise ein „true“ zurück, was jedoch falsch ist, da 1 und 5 natürlich weiter auseinander liegen als der Abstand 3.
Daraus folgt, dass auch die letzte Methode nicht zwangsweise zu Testdaten führt, die alle Fehler findet. Allerdings ist sie allemal zufälligen Testdaten vorzuziehen.
Hieraus wird jedoch ersichtlich, dass weitere Methoden benötigt werden, die zu guten Testdaten führen. Im nächsten Kapitel werden weitere Methoden vorgestellt, die jedoch unter anderen Gesichtspunkten ausgeführt werden und zwar mit Hilfe des Blackbox-Tests.

Blackbox-Test

Beim Blackbox-Test ist, wie oben schon erwähnt, der Quellcode eines Programms nicht bekannt. Hierbei analysiert man ausschließlich die Spezifikation des Programms unter Berücksichtigung der Ein- und Ausgabedaten und versucht auch hier mit verschiedenen Methoden, die man zur Hand hat, gute Testdaten zu finden.
Zuallererst soll gezeigt werden, dass vollständiges (erschöpfendes) Testen auch hierbei unmöglich ist. Hierzu betrachten wir das letzte Beispiel aus dem folgenden Kapitel, bei dem mit dem fehlerhaften Programm „If (a – b < c) …;“ untersucht werden soll, ob zwei Zahlen kleiner als der Abstand c ist. Beschränkt man dabei noch alle Eingabezahlen a, b und c auf ganze positive Zahlen und will man das Programm unter Berücksichtigung des Backbox-Testens vollständig testen, so müsste man alle Werte bis zum Maximalwert der ganzen Zahlen (int) für a, b und c einsetzen und alle möglichen Kombinationen damit testen. Ich erspare mir jetzt einmal die Berechnung, wie lange man dafür brauchen würde, aber ich denke es wird für jeden deutlich, dass auch diese Zahl exorbitant hoch wäre und man diese Tests nicht alle durchführen könnte. Also braucht man auch hier Methoden, um brauchbare Testdaten zu finden. Dies kann man z.B. durch die Bildung von Äquivalenzklassen erreichen.

Äquivalenzklassen

Die erste Methode ist die Bildung von Äquivalenzklassen. Dabei versucht man, Mengen voraussichtlich gleichartiger Testdaten zu einer Äquivalenzklasse zusammenzufassen, mit der Schlussfolgerung, dass wenn man einen Testfall aus dieser Äquivalenzklasse definiert, sich alle anderen Testfälle aus dieser Äquivalenzklasse voraussichtlich gleich verhalten werden. Die Schwierigkeit dabei ist, geeignete Äquivalenzklassen zu finden. Dies kann schlecht automatisiert werden, es ist – wie auch andere Blackbox-Methoden – ein größtenteils intuitiver Prozess; es gibt jedoch ein paar Richtlinien bzw. Hilfen, die einem dabei helfen können. Hier seien einmal ein paar erwähnt:

1) Beachtung des Wertebereichs bei Eingaben. Soll z.B. bei einem Programm der Eingabewert X zwischen 10 und 20 liegen, so erstellt man eine Äquivalenzklasse mit „normalen“ gültigen Werten, eine Äquivalenzklasse mit zu niedrigen Werten und eine mit zu hohen Werten.
2) Kann es eine verschiedene Anzahl von Eingaben geben? Hat man beispielsweise bei einem Bibliothekssystem die Möglichkeit für ein Buch mindestens 1, aber max. 20 Autoren anzugeben, so definiert man eine Äquivalenzklasse, die 1 bis 20 Autoren umfasst, eine Äquivalenzklasse mit mehr als 20 Autoren und eine Äquivalenzklasse mit keinem Autor.
3) Gibt es festgelegte Werte bei Eingaben, z.B. kann man bei einem Erfassungssystem innerhalb einer Hochschule einer Person vordefinierte Zustände zuweisen wie z.B. Student, Professor, Mitarbeiter, etc. und erwartet man, dass diese sich vermutlich verschieden verhalten, so definiert man pro gültigem Wert eine Äquivalenzklasse (also eine für Student, eine für Professor, etc) und eine Äquivalenzklasse für einen ungültigen Wert (z.B. Taxifahrer).
4) Usw.

Mit dieser Methode definiert man seine Äquivalenzklassen für gültige und ungültige Werte. Auf diesen baut die weitere Vorgehensweise zur Findung von Testfällen auf. Das generelle Vorgehen sieht also wie folgt aus:

1) Definition der Äquivalenzklassen.
2) Testfälle finden, die so viele gültige Äquivalenzklassen wie möglich abdecken.

3) Testfälle erzeugen, die je 1 (und genau 1) ungültige Äquivalenzklasse abdecken.

Bei Punkt 3) ist wichtig, dass man nur einen Testfall pro ungültige Äquivalenzklasse definiert, da man bei diesem Testfall einen Fehler erwartet. Erstellt man nun Testfälle, die mehrere ungültige Äquivalenzklassen umfassen, so maskieren diese sich gegenseitig, da in der Regel nur einer der Fehler gemeldet wird und das Programm danach nicht bis zum zweiten Fehler weiterarbeitet.

Diese Vorgehensweise soll verhindern, dass unsinnig viele ähnliche Testfälle erstellt werden, die keine qualitativ sinnvolle Erweiterung des Testverlaufs darstellen.

Grenzwertanalyse

Als weiteres Blackbox-Verfahren gibt es die so genannte Grenzwertanalyse. Dabei wird der Schwerpunkt auf die Grenzwerte der Ein- und Ausgabedaten gelegt, da oftmals an den „Rändern“ viele Fehler auftreten. Falsche „if“-Abfragen wie z.B. $X > 1$ statt $X \geq 1$ können so sehr gut aufgedeckt werden. Hier kann man sehr gut die im vorherigen Unterkapitel behandelten Äquivalenzklassen benutzen. Man betrachtet nun nämlich genau die Grenzwerte dieser Äquivalenzklassen und definiert genau für die „Grenzwerte“ seine Testfälle.

Für das Beispiel der Bibliothekssoftware, die 1 bis 20 Autoren pro Buch erfassen soll, würden also Testfälle für

- keinen Autor
- 1 Autor
- 20 Autoren
- 21 Autoren

erstellt werden. Wie man sieht, werden genau die Grenzwerte behandelt. 1 Autor und 20 Autoren sind gerade noch zulässig, wobei kein Autor und 21 Autoren gerade so nicht mehr gültig sind. Sie liegen also exakt auf und um die Grenzwerte herum.

Wichtig ist auch, dass man „Ausgabe-Äquivalenzklassen“ nicht vergisst. Dies soll anhand eines Beispiels verdeutlicht werden: Berechnet ein Programm anhand des Umsatzes von Angestellten die jeweilige Provision. Diese soll zwischen 0 und 1000 Euro liegen. Hier versucht man nun Eingabedaten zu finden, die genau 0 und genau 1000 Euro ergeben. Außerdem ist zu analysieren, ob man mit bestimmten Eingaben diese Werte evtl. über- oder

unterbieten könnte, man also entweder eine negative Provision oder eine Provision > 1000 Euro erreichen kann.
Vor allem bei den „ungültigen“ Ausgabewerten ist auch bei dieser Methode Intuition gefragt; bei den Eingabewerten kann man sich andererseits sehr gut an den vorher spezifizierten Äquivalenzklassen orientieren.

Ursache-Wirkungs-Graph

Eine Methode, die strukturierter an die Sache herangeht ist der Ursache-Wirkungs-Graph. Im Gegensatz zu den bisher besprochenen Verfahren (der Grenzwertanalyse und der Bildung von Äquivalenzklassen) berücksichtigt dieses System Kombinationen von Eingabewerten. Außerdem wird hier die logische Struktur eines Programms untersucht (ohne Kenntnis des Quelltextes ==> kein Whitebox-Test) und mithilfe des Ursache-Wirkungs-Graphen zur Findung von Testfällen verwendet.
Da dieses Verfahren sehr schnell sehr komplex werden kann, bietet es sich an, die Spezifikation eines Programms in mehrere „handliche Stücke“ zu unterteilen, da die Abwicklung dieser Methode sonst zu umfangreich und komplex werden würde. Als Beispiel sei hier genannt, dass wenn man einen Compiler testen möchte, man jede Anweisung als Individuum behandeln könnte und man für jede Anweisung einen eigenen Ursache-Wirkungs-Graphen erstellen könnte.
Zu allererst einmal gibt es wie der Name schon sagt Ursachen und Wirkungen. Ursachen sind die Eingangsbedingungen und die Wirkungen sind Ausgangsbedingungen oder Systemtransformationen. Unter Systemtransformationen könnte man beispielsweise die Veränderung einer Datei verstehen. Diese Ursachen und Wirkungen entnimmt man der Spezifikation und erstellt einen boolschen Graphen, der Ursachen und Wirkungen miteinander in Verbindung setzt. Dies sei an dieser Stelle anhand eines Beispiels klarer gemacht. In Abb.10 sehen wir den Auszug aus einer Spezifikation. Es handelt sich hierbei um das Auslesen aus einer Datei. Analysiert man diese Spezifikation, so ergeben sich folgende Ursachen und Wirkungen.

Beispiel:

Das Zeichen in Spalte 1 muss A oder B sein. Das Zeichen in Spalte 2 muss eine Ziffer sein. Unter diesen Umständen wird die Ergänzung einer Datei durchgeführt. Ist das erste Zeichen nicht korrekt, so wird die Meldung X 12 ausgegeben. Ist das zweite Zeichen keine Ziffer, so wird die Meldung X 13 ausgegeben.

Abbildung 10: Auszug aus einer Spezifikation

Ursachen:

1) Zeichen in Spalte 1 = A.
2) Zeichen in Spalte 1 = B.
3) Zeichen in Spalte 2 ist eine Ziffer.

Wirkungen:

70) Ergänzung wird durchgeführt
71) X 12 ausgeben.
72) X 13 ausgeben.

Diese Ursachen und Wirkungen setzt man nun unter Berücksichtigung der logischen Symbole (Abb.11) und Einschränkungen (Abb.12) in Beziehung und erhält dadurch den Ursache-Wirkungs-Graphen. Doch zuerst ein paar Erklärungen zu den logischen Symbolen in Abb.11 und was sie bedeuten:

- Identity: „a“ und „b“ sind immer identisch, also a = b.
- NOT: Wenn a = 0 <==> b = 1 und a =1 <==> b = 0.
- OR: Ist mindestens eines aus „a“, „b“ oder „c“ = 1, so ist d = 1, sonst ist d = 0.
- AND: „a“ und „b“ müssen beide den Wert 1 besitzen, um auch bei „c“ eine 1 zu erhalten.

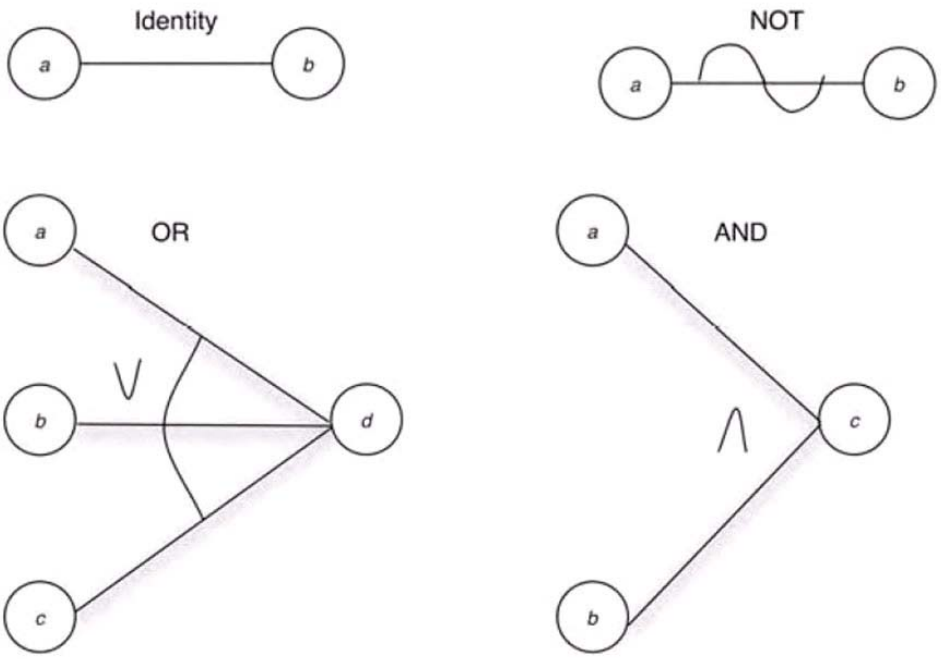

Abbildung 11: Logische Symbole

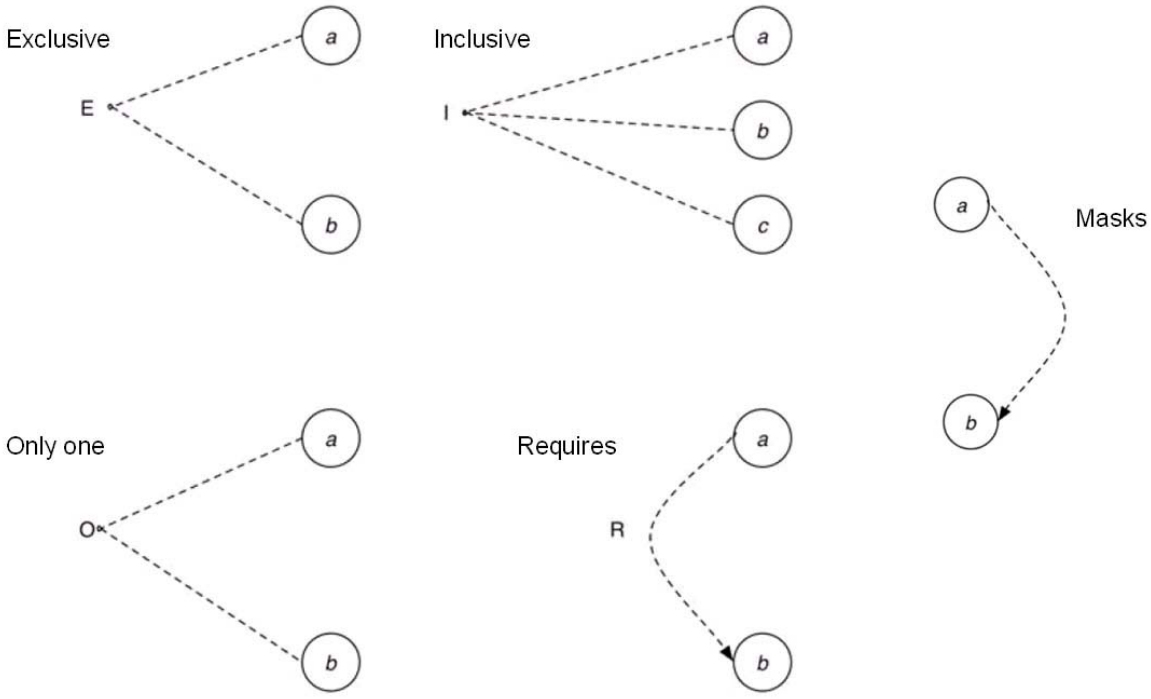

Abbildung 12: Einschränkungen

Außerdem gibt es noch die Einschränkungen, die in Abb.12 zu sehen sind:

- E (Exclusive): Höchstens eins der beiden darf 1 sein.
- I (Inclusive): Mindestens eins der drei muss 1 sein.
- O (Only one): Genau ein Zustand muss 1 sein.
- R (Required): Ist a = 1, so ist auch b = 1, was jedoch nicht (!) heisst, dass bei a = 0 auch b = 0 folgt.
- M (Masks): Ist a = 1, so ist b = 0 („a“ maskiert „b“). a = 0 hat keinen Effekt auf b.

Mit Hilfe dieser Symbole und Einschränkungen lässt sich aus dem vorherigen Beispiel folgender Ursache-Wirkungs-Graph erstellen:

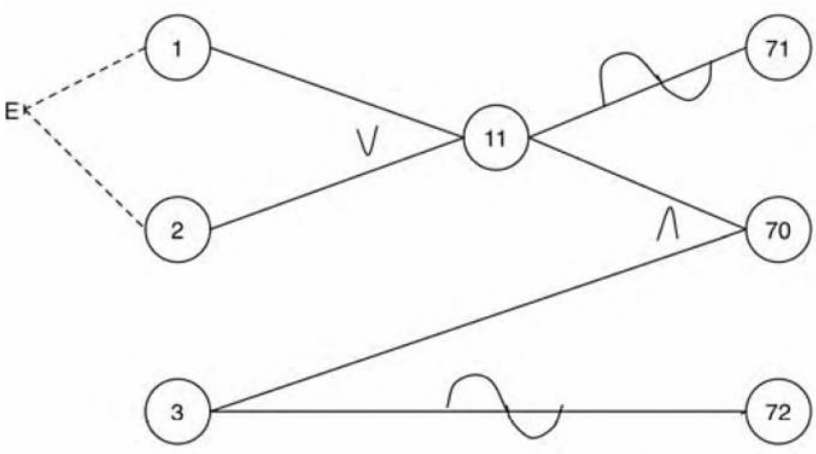

Abbildung 13: Ursache-Wirkungs-Graph des Beispiels

Links stehen die Ursachen 1 – 3 und rechts stehen die Wirkungen 70 – 72. Wie man hier sieht, wurde der Zwischenzustand 11 erzeugt. Dieser bedeutet soviel wie „Zeichen in Spalte 1 ist ein gültiges Zeichen“ (nämlich entweder A oder B).
Der nächste Schritt besteht nun darin, diesen Ursache-Wirkungs-Graphen in eine Entscheidungstabelle umzusetzen. Dabei werden die Wirkungen (70 – 72) nacheinander auf 1 gesetzt und der Graph bis zu den Ursachen zurückverfolgt. Hier muss man nicht jede mögliche Kombination berücksichtigen – es gibt gewisse Richtlinien, die die Anzahl der Ergebnisse einschränken. Die Entscheidungstabelle für das bisher behandelte Beispiel würde, wie in Abb.14 zu sehen ist, anfangen.

Ursache / Wirkungen	T1	T2	...
1	1	0	...
2	0	0	...
3	1	1	...
70	1	0	...
71	0	1	...
72	0	0	...

Abbildung 14: Entscheidungstabelle

Aus dieser Entscheidungstabelle lassen sich nun direkt aus den Spalten eindeutige Testfälle ableiten. T1, T2, etc. stehen für die einzelnen Testfälle. Darunter kann man ablesen, was für Ursachen und was für Wirkungen dabei einzusetzen / zu erwarten sind:

- T1: Eingabe aus Datei = „A5“; erwartete Ausgabe = Ergänzung wird durchgeführt.
- T2: Eingabe aus Datei = 35“; erwartete Ausgabe = X 12 wird ausgegeben.

Bei T1 beispielsweise ist aus der Entscheidungstabelle abzulesen, dass Ursache 1) erfüllt ist, also muss in der Datei in der ersten Spalte ein „A“ stehen. Außerdem ist Ursache 3) erfüllt, woraus folgt, dass in der zweiten Spalte eine Ziffer zu stehen hat. Als Wirkung erwartet man die 71), nämlich, dass die Ergänzung durchgeführt wird.

Dieses Vorgehen sah bisher sehr einfach aus, doch hier handelt es sich ja auch nur um ein Trivialbeispiel. Diese Ursache-Wirkungs-Graphen können sehr schnell sehr komplex werden. Dies ist in Abb.15 zu sehen.
Wie man sieht, ist dieser Ursache-Wirkungs-Graph schon deutlich komplexer. Die Schwierigkeit, die sich dabei ergeben wird, ist, aus diesem Graphen eine Entscheidungstabelle zu erstellen. Dies könnte man jedoch mit Hilfe eines Programms sehr einfach automatisieren. Ein sehr großer Vorteil dieser Methode liegt darin, dass die logischen Zusammenhänge eines Programms sichtbar werden und Fehler in der logischen Struktur so sehr gut aufgedeckt werden können.

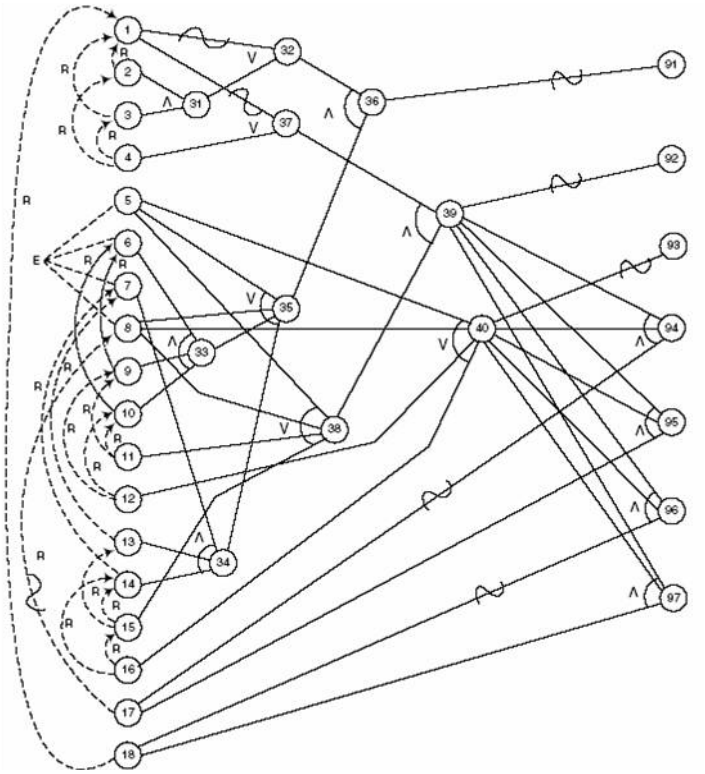

Abbildung 15: Ursache-Wirkungs-Graph

Fehlererwartung

Nun kommen wir zum letzten Punkt bei den Backbox-Tests - der Methode der Fehlererwartung. Dabei wird abgeschätzt, wo Fehler in dem Programm bzw. beim Programmieren aufgetreten sein könnten. Dabei könnte man beispielsweise Testfälle entwerfen, die bei der Eingabe eine Null enthalten oder eine leere Eingabedatei. Auch das Verhalten bei doppelten gleichen Eingaben, etc. eignet sich hier gut zum Untersuchen – alle Fälle also, an die der Programmierer evtl. nicht gedacht haben könnte. Dies ist meiner Meinung nach der intuitivste Prozess in Bezug auf den Entwurf von Testfällen. Auch wenn versucht wird Statistiken zu erstellen, wo die meisten üblichen Fehler zu erwarten sind, wird man diese Methode doch nicht ganz automatisieren können und es wird immer ein bisschen Intuition gefragt sein.

Der zweite Punkt der Fehlererwartung ist derjenige, dass sich in der Praxis gezeigt hat, dass Fehler meist nicht alleine auftreten. Findet man also einen Fehler in einem bestimmten Teil eines Programms, so ist die Wahrscheinlichkeit hoch, hier auch noch weitere Fehler zu finden. Warum dies so ist, kann nicht genau erklärt werden, aber ich könnte mir denken, dass

vor allem Fehler einerseits bei komplexen Problemstellungen auftreten, bei denen die Wahrscheinlichkeit hoch liegt, dass sich durch die Komplexität noch weitere Fehler eingeschlichen haben. Auch könnte es z.B. daran liegen, dass der Programmierer einen bestimmten Code-Abschnitt am Ende eines Arbeitstages entworfen hat zu einem Zeitpunkt, zu dem er schon ziemlich müde war und dass sich somit bei ihm mehr Fehler eingeschlichen haben. Fakt ist, dass man Programm-Abschnitte, bei denen Fehler auftreten, noch einmal genauer „unter die Lupe“ nehmen sollte.

Vorgehensweise bei der Testplanung

Im ersten Abschnitt dieses Seminars wurden also verschiedene Methoden zur Findung von Testfällen vorgestellt. Nun stellt sich aber die Frage: Wie geht man also bei der Testplanung vor, welche Methode ist die Beste zum Auffinden von Fehlern?

Darauf gibt es nur eine Antwort: Es gibt nicht „die“ beste Methode. Geht man nun konkret daran, ein Programm zu testen, so sollte man das Programm erst mit einer der Whitebox-Methoden analysieren, wobei die „Ausführung aller Mehrfachbedingungen“ sicherlich die sinnvollste, aber auch komplexeste Methode dabei ist. Es macht durchaus Sinn in gewissen Situationen auf eine weniger komplexe Vorgehensweise beim Whitebox-Test auszuweichen. Natürlich wird dadurch die Wahrscheinlichkeit größer, qualitativ weniger gut geeignete Testdaten zu erhalten, doch oftmals hat man einfach nicht die Zeit für die Ausführlichste Testmethode. Mit dem Whitebox-Test ist es jedoch alleine nicht getan. Im Weiteren sollte man möglichst alle hier vorgestellten Blackbox-Methoden heranziehen, wenn man seine Software gründlich testen möchte, denn jede Methode hat ihre Stärken. Die Grenzwertanalyse beispielsweise zielt auf bestimmte Fehlerquellen an den Rändern der Spezifikationseingabe- und Ausgabewerten hin, wobei man mit dem Ursache-Wirkungs-Graphen die logische Struktur des Programms unter die Lupe nimmt. Daher reicht es nicht, nur eine Methode zum Testen zu verwenden.

Man sollte jedoch immer im Kopf behalten, dass selbst durch das Verwenden des besten Whitebox-Tests und aller hier vorgestellten Blackbox-Methoden keine komplette Fehlerfreiheit garantiert werden kann, aber es besteht dadurch eine große Wahrscheinlichkeit, qualitativ gute Testdaten zu finden, mit denen man viele Fehler aufdecken kann.

Modultests

Kommen wir nun zu dem Abschnitt der Modultests. In Abb.16 sieht man ein aus Modulen (A bis L) zusammengesetztes Programm. Die Eingabe findet in Modul J statt, die Ausgabe läuft über I. Das Diagramm ist so zu verstehen, dass Modul A die Module B, C und D aufruft. Modul B wiederum benutzt E und F, usw.

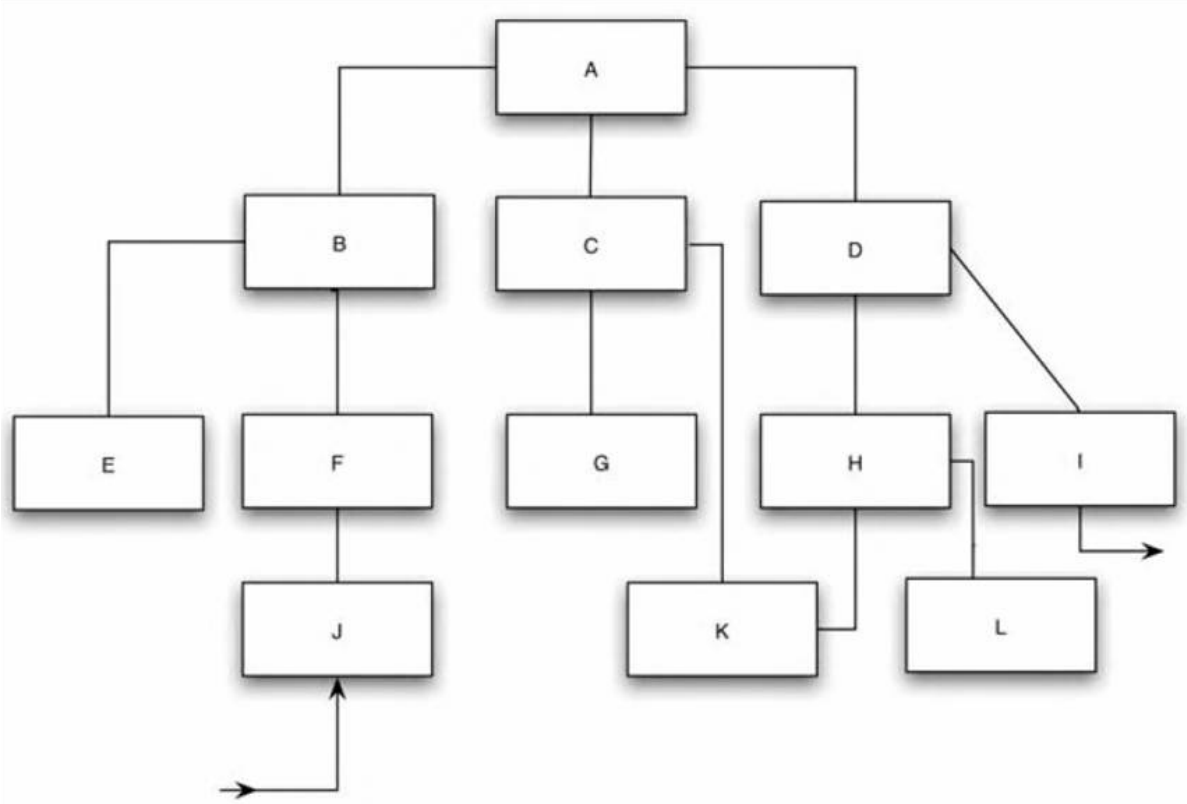

Abbildung 16: Modulansicht eines Programms

Möchte man ein Programm, welches aus vielen Modulen zusammengesetzt ist, testen, so bietet es sich an, die Whitebox- und Blackbox-Methoden auf die einzelnen Module anzuwenden. Dadurch wird der Testvorgang auf überschaulichere, weniger komplexe Programmabschnitte angewandt. Hierbei gibt es aber mehr zu beachten als man auf den ersten Blick erwarten würde. Es gibt hierbei verschiedene Vorgehensweisen, wie man die einzelnen Module testen kann.

Nichtinkrementelles Testen

Die erste Methode, die einem in den Sinn kommen mag, ist, dass man jedes Modul einzeln testet und danach alle Module zu dem fertigen Programm zusammenfügt. Dadurch kann man alle Module gleichzeitig (parallel) überprüfen – sofern man genug Tester zur Verfügung hat. Dies ist das so genannte „nichtinkrementelle Testen“. Doch schon hier stößt man auf die ersten Schwierigkeiten.

Möchte man beispielsweise Modul B testen, so bekommt man das Problem, dass dieses die Module E und F benötigt, da diese in Modul B aufgerufen und verwendet werden. Allerdings möchte man nur Modul B testen und nicht schon gleich E und F. Daher müssen diese „simuliert“ werden. Dies geschieht mit so genannten STUBs, welche die erwarteten Rückgabewerte von E und F zurückliefern. Außerdem benötigt man zusätzlich einen Testtreiber, welcher Modul B mit verschiedenen Testdaten aufruft (Abb.17).

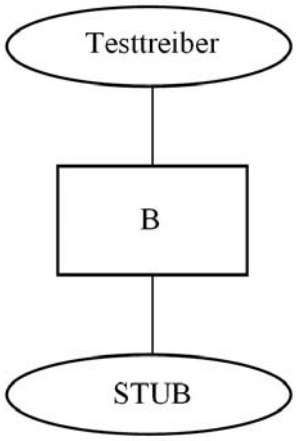

Abbildung 17: Aufruf eines Moduls durch einen Testtreiber unter Verwendung von STUBs

Sind Testtreiber noch recht einfach umzusetzen, so wird es bei dem Entwurf von STUBs um einiges komplizierter. Hat beispielsweise Modul E die Aufgabe, die Wurzel einer Zahl auszurechnen und ruft Modul B dieses bei einem Durchlauf häufig auf (z.B. 500-mal), so muss man diese 500 Werte einzeln ausrechnen und in das STUB eingeben, damit Modul B dann die richtigen Werte beim Testlauf zur Verfügung hat. Bei mehreren verschiedenen Testfällen wird das dadurch extrem viel Arbeit. Leider führt daran jedoch bei dieser Methode kein Weg vorbei, da man die Wurzelfunktion ja nicht einfach programmieren kann, denn

somit hätte man Modul E „nachprogrammiert“ mit wiederum möglichen Fehlerquellen. Man sieht also, STUBs zu entwerfen ist keineswegs trivial.
Ein weiterer Nachteil bei dieser Methode ist, dass man die Module erst nach dem kompletten Test aller einzelnen Module zusammensetzt. Eventuelle Probleme, die aufgrund von unklaren Spezifikationsdefinitionen und somit erst beim Zusammenspiel verschiedener Module auftreten könnten, werden nun erst bei dem komplett zusammengesetzten Programm in Erscheinung treten. Dabei wird es nun unter Umständen schwierig, die Fehlerquelle zu lokalisieren. Der Vorteil des Modultestens – des Testens einzelner kleiner überschaubarer Programmteile – ist verloren.
Dies bringt uns zur nächsten Vorgehensweise, dem „Inkrementellen Testen“.

Inkrementelles Testen

Bei dieser Methode werden die Module nicht mehr alle unabhängig voneinander getestet, sondern man fängt entweder ganz oben (Top-Down-Testen) oder ganz unten (Bottom-Up-Testen) an und testet im ersten Schritt diese Module. Im weiteren Verlauf geht man dann jeweils einen Schritt nach unten (bzw. oben) weiter und nimmt sich die nächsten Module vor, wobei man nun aber die schon getesteten Module direkt verwendet.
Startet man den Test beispielsweise bei Modul J (siehe Abb.16), so wird man als nächstes weiter nach oben vorgehen und Modul F testen, wobei man J nicht durch STUBs simuliert, sondern man das bereits getestete Modul direkt verwendet. Dadurch können die Module zwar nicht mehr parallel getestet werden, aber es ergeben sich hieraus auch viele Vorteile, die meiner Meinung nach deutlich überwiegen:

- Man benötigt nur noch Testtreiber oder STUBs, aber nicht mehr beides gleichzeitig.
- Schnittstellenfehler werden direkt beim Testen des nächsten Moduls sichtbar und sind somit leicht lokalisierbar.
- Die Module werden indirekt mehrfach getestet. Das erste mal speziell bei den auf das Modul zugeschnittenen Testläufen und danach noch einmal indirekt, wenn diese Module im weiteren Test verwendet werden.

Wie man hier sehen kann, überwiegen deutlich die Vorteile dieses inkrementellen Testens. Aber auch hier gibt es, wie oben schon erwähnt, zwei verschiedene Ansätze, das Top-Down-Testen und das Bottom-Up-Testen.

Top-Down-Testen

Beim Top-Down-Testen beginnt man den Test an der Spitze des Programms. Bei Abb.16 wäre das mit Modul A. Im weiteren Verlauf des Tests arbeitet man sich dann weiter nach unten durch unter Weiterverwendung von Modul A.

Hierbei werden zwar keine Testtreiber verwendet, sondern nur STUBs, aber diese sind, wie vorhin schon erwähnt, teilweise nicht trivial umzusetzen. Außerdem besitzt das oberste Modul meist keine eigene Ein- bzw. Ausgabe, sondern dies wird durch andere Module ausgeführt. In Abb.16 beispielsweise ist Modul J für die Eingabe zuständig und Modul I für die Ausgabe. Daraus folgt, dass die Eingabe der Testdaten, sowie die Auswertung des Tests auch über STUBs umgesetzt werden müssen. Dieses Phänomen verkompliziert noch einmal mehr das Testen mithilfe von STUBs.

Nachdem man Modul A getestet hat, ist es im Grunde egal, welches Modul als nächstes „angeschlossen“ und getestet wird, allerdings gibt es sinnvolle Richtlinien, die man möglichst beachten sollte:

1) Kritische Module, die voraussichtlich eher fehleranfällig sind, sollten möglichst früh eingebunden werden.
2) Eingabe-/Ausgabemodule möglichst früh einbinden. Dies hat den Vorteil, dass man die Ein- und Ausgabe der Testdaten nun nicht mehr über STUBs realisieren muss, sondern man mit realen Ein- und Ausgabewerten arbeiten kann.

Ein zu erwähnender weiterer Vorteil dieses Modells ist es, dass man, sobald die E/A-Module eingebunden wurden, ein ausführbares Programmskelett besitzt. Dabei werden zwar viele Teile des Programms noch über STUBs realisiert, aber man sieht zumindest schon mal das fertige laufende Programm. Dies kann sehr motivationsfördernd sein, was nicht zu unterschätzen ist.

Andererseits tritt hier ein anderes Problem zutage. Betrachten wir das obere Diagramm (Abb.16) und nehmen wir an, dass Modul G getestet werden soll. Die Ein- und Ausgabemodule sind schon integriert, so dass man über diese die Testdaten eingeben, sowie auswerten kann. Nun werden diese Testdaten auf dem Weg zu Modul G allerdings schon verändert und es kann kompliziert, ja sogar unmöglich werden, gewisse Testdaten bis zu Modul G vordringen zu lassen. Beim Testen von Modul G möchte man beispielsweise Testdaten, welche aus der Grenzwertanalyse gewonnen wurden, anlegen, welche genau am

Rand außerhalb der Spezifikation liegen, um so die Fehlerbehandlung zu überprüfen. Nun ist es wahrscheinlich, dass die Eingabedaten, die dazu verwendet werden, schon in einem Modul, welches sie vorher durchlaufen, eine Fehlermeldung erzeugen. Oder es ist vielleicht sogar unmöglich, über diese vorherigen Module den Testwert zu erzeugen, der dann bei Modul G anliegen soll. Dies ist ein weiteres schwerwiegendes Problem, welches beim Top-Down-Testen zutage tritt. Außerdem werden die auszuwertenden Rückgabewerte von Modul G bis zum Ausgabemodul I meist weiter verändert, so dass man auch diese Rückgabewerte nur indirekt zu sehen bekommt.
Wie man sieht, treten auch hier beim Top-Down-Testen viele Schwierigkeiten und Probleme auf. Das bringt uns zur nächsten Vorgehensweise, dem Bottom-Up-Testen.

Bottom-Up-Testen

Beim Bottom-Up-Testen arbeitet man sich im Gegensatz zum Top-Down-Testen nicht von oben nach unten, sondern man beginnt bei den Modulen auf der untersten Ebene (Abb.16) und arbeitet sich nach oben durch. Beispielsweise beginnt man mit dem Modul J. Hat man dieses ausführlich getestet, so nimmt man sich als nächstes Modul F unter Verwendung von Modul J vor, usw. Wie man gleich sehen wird, werden die Nachteile der Top-Down-Methode zu Vorteilen der Bottom-Up-Methode und die Vorteile werden zu Nachteilen. Daher sei das Ganze hier nur noch einmal relativ kurz erläutert:

- Bei dieser Methode umgeht man das komplizierte Umsetzen von STUBs, da man diese hier nicht braucht. Dafür muss man allerdings für jedes Modul Testtreiber entwerfen, die diese Module aufrufen. Testtreiber sind jedoch in der Regel leichter umzusetzen als STUBs.
- Hier wird auch gleich ein zweiter Vorteil sichtbar. Man ruft die Module direkt durch die Testtreiber auf und hat so einen direkteren Zugriff auf die Module. Daher kann man hier meist auch direkt die gewollten ungültigen Eingaben (Grenzwertanalyse) recht einfach umsetzen. Auch die Rückgabe von Testdaten ist oft einfacher und somit leichter auswertbar.

- Allerdings gibt es auch hier leider einen Nachteil. Bis man das letzte Modul getestet hat, wird man kein laufendes Programmskelett vorfinden. Erst wenn (Abb.16) Modul

A eingebunden wurde, hat man ein lauffähiges Programm – und dieses Programm ist dann schon das endgültige Programm. Sollten hier plötzlich Designfehler auffallen, so wird es sehr teuer diese zu korrigieren, da diese Fehler erst sehr spät (nämlich nach Beendigung des Tests) aufgefallen sind.

Trotzdem bin ich der Meinung, dass die Vorteile des Bottom-Up-Testens deutlich überwiegen und man beim Testen von Modulen meiner Meinung nach möglichst diese Methode anwenden sollte, da sie am einfachsten und effektivsten umgesetzt werden kann und damit voraussichtlich die besten Ergebnisse in akzeptabler Zeit erzielt werden.

Schlusswort

Ich hoffe, dass dieses Dokument einen Überblick vermitteln konnte, wie man strukturiert sinnvolle Testdaten finden kann und was speziell bei Modultests zu beachten ist. So dass Sie dann bei Ihrem nächsten Testen (im Studium oder im Beruf) hieraus einige Hilfen und Vorgehensweisen mitnehmen konnten, sich dann wieder daran erinnern und dadurch strukturiert an die Sache herangehen können. Viel Erfolg beim Testen.
Tim Aßmann

Literaturverzeichnis

1) G. J. Myers, „Methodisches Testen von Programmen“, 7. Auflage, 2001
2) Boris Beizer, „Black-Box Testing“, 1995

Abb. 1 – 3, 8, 10 – 13, sowie 15 und 16 sind aus 1) G. J. Myers, „Methodisches Testen von Programmen“, 7. Auflage, 2001 übernommen.